Efficient Spatial Joins on Large Geometry Sets

Hannah Bast
University of Freiburg
Freiburg, Germany
bast@cs.uni-freiburg.de

Patrick Brosi
University of Freiburg
Freiburg, Germany
brosi@cs.uni-freiburg.de

Johannes Kalmbach
University of Freiburg
Freiburg, Germany
kalmbach@cs.uni-freiburg.de

Abstract

We consider the following standard spatial-join problem: Given two sets of geometric objects in 2D (points, lines, polygonal areas, and collections of these), compute the spatial relations of all pairs of intersecting objects as a standard DE-9IM matrix. Most previous work focuses on one aspect of the problem, like the candidate generation, candidate reduction heuristics, efficient data structures, or parallelization. We provide a complete, fully functional, and carefully engineered implementation, as well as an extensive experimental evaluation of the relevance of various heuristics and of two variants for the exact geometry comparisons: our own implementation which preprocesses the geometries, and one using the *GEOS* library, which powers spatial joins in the widely used PostgreSQL+PostGIS. In particular, we find that the former speeds up spatial joins by more than an order of magnitude when complex geometries are involved. Our best approach can compute the full self join of the 1.4 billion geometries from OpenStreeMap in less than 3 hours on a commodity PC. This was out of reach for any existing implementation we tried. Our code and all the materials needed to reproduce our results are freely available on GitHub.

CCS Concepts

• **Theory of computation → Computational geometry**; • **Information systems → Geographic information systems**.

Keywords

Spatial Joins, DE-9IM, Spatial Databases, OpenStreetMap

ACM Reference Format:
Hannah Bast, Patrick Brosi, and Johannes Kalmbach. 2025. Efficient Spatial Joins on Large Geometry Sets. In *The 33rd ACM International Conference on Advances in Geographic Information Systems (SIGSPATIAL '25), November 3–6, 2025, Minneapolis, MN, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3748636.3762757

1 Introduction

The computation of spatial relations between geometric objects is a core functionality of spatial libraries and databases. Standard spatial relations are *contains*, *covers*, *within*, *intersects*, *touches*, *crosses*, and *equals* [13]. For example, consider the following queries on the OpenStreetMap data, which as of this writing contains around 1.4 billion geospatial objects:

All streets in Germany
For each of a large set of regions, all contained streets
All street crossings

Spatial joins are typically computed in three (often interleaving) stages. The first stage generates all pairs of objects, where the axis-parallel bounding boxes intersect (other pairs cannot be part of the result, see the definition below for details). The second stage reduces this set of candidate pairs via heuristics (for example, via more tightly fitting bounding boxes). The third stage computes the exact spatial relations for each remaining candidate pair.

For example, the widely used PosgreSQL+PostGIS system realizes the first stage by looking up each object from the smaller of the two input sets in an R-tree-like index structure precomputed for the larger of the two input sets. No filtering heuristics are used, and exact geometric computations are done using the *GEOS* library.

In this paper, we use a sweep line approach for the first stage, and investigate a number of variations for the second and third stage. For the second stage, we consider five different heuristics (three of which are tighter bounding polygons). For the third stage, we consider two variants: one based on preprocessed geometries and one using the *GEOS* library. Our findings regarding the second and third stage are not specific for the sweep line approach.

1.1 Problem Definition

We consider the problem of computing the *spatial join* between two sets O_1 and O_2 of geometric objects in 2D.

Each object can be either a point, a polygonal line, an area that is delineated by one or several polygonal lines[1], or arbitrary collections of these. Specifically, we assume that each object is given as a pair of an ID (an arbitrary string, unique for that object) and a Well-Known Text (WKT) string defining the geometry. For example, the object for "Freiburg main station" could be given as: *osmnode:21769883 POINT(7.8412948 47.9977308)*.

The spatial relationship between two geometric objects o_1 and o_2 can be expressed by their DE-9IM matrix [8, 13]. It is defined as follows, where I, E and B map to 2D point sets and $I(o)$ is the interior of o (e.g. the inside of a polygon), $B(o)$ is the boundary of o (e.g. the border of a polygon), and $E(o)$ is the exterior of o (everything not belonging to the interior or the boundary):

$$\begin{bmatrix} \dim(I(o_1) \cap I(o_2)) & \dim(I(o_1) \cap B(o_2)) & \dim(I(o_1) \cap E(o_2)) \\ \dim(B(o_1) \cap I(o_2)) & \dim(B(o_1) \cap B(o_2)) & \dim(B(o_1) \cap E(o_2)) \\ \dim(E(o_1) \cap I(o_2)) & \dim(E(o_1) \cap B(o_2)) & \dim(E(o_1) \cap E(o_2)) \end{bmatrix}$$

The standard relations *contains*, *covers*, *within*, *intersects*, *touches*, *crosses*, and *equals* can be easily computed from the DE-9IM matrix. Note, however, that the set of spatial relationships captured by a DE-9IM matrix is bigger than these standard relations.

[1] In particular, areas can consist of multiple rings, with holes, and then again areas insides the holes, etc.

SIGSPATIAL '25, Minneapolis, MN, USA

ACM ISBN 979-8-4007-2086-4/2025/11
https://doi.org/10.1145/3748636.3762757

This matrix is often serialized as a 9-character string, where each character is 0, 1, or 2 if the respective intersection has that dimension, or F if the intersection is empty. For more details, especially regarding the definitions of $I(o)$, $B(o)$ and $E(o)$ for points and lines, we refer to [13].

Our goal is to compute the DE-9IM matrix of all pairs (o_1, o_2), with $o_1 \in O_1$ and $o_2 \in O_2$, for which o_1 and o_2 intersect (for non-intersecting geometries, the DE-9IM matrix is trivial). We output this information as a list of triples. For example, for the two objects "Germany" and "Freiburg main station" we output *osmrel:51477 0F2FF1FF2 osmnode:21769883*.

1.2 Contributions

We provide a complete, fully functional, and carefully engineered implementation for computing spatial joins according to the definition above. A detailed description is provided in Section 3. We do not claim novelty for any of the individual parts. What distinguishes our work are two aspects:

- Most previous work focuses on one particular aspect of the problem, like the candidate generation, heuristics for reducing the number of candidate pairs, suitable data structures, parallelization, or special hardware. We consider all aspects relevant for a complete implementation. It is highly non-trivial to understand which of the many ideas from the vast literature actually work well in practice, and to then implement them efficiently.
- We provide a detailed description of our algorithm, and a complete and fully functional implementation as free open-source software. There are only very few such implementations because previous work either focuses on particular aspects (see the previous point), has not been implemented at all, or the implementation is dysfunctional or not publicly available.

We provide an extensive experimental evaluation of our approach and many of its variants, see Section 4. In particular:

- We evaluate two variants for the exact geometric comparisons: our own implementation based on preprocessed geometries, and the implementation of the widely used *GEOS* library, which in particular powers PostgreSQL+PostGIS. It turns out that our own implementation outperforms *GEOS* by at least an order of magnitude for joins involving complex geometries, while not being slower for joins involving simpler geometries.
- We evaluate the effect of a variety of heuristics for reducing the set of candidate pairs, with interesting results. In particular, there is one heuristic (*cell IDs*, see Section 3.7) that turns out to be crucial for both of our variants of the exact geometry comparisons, while there is another heuristic (*inner-outer approximations*, see Section 3.6.3) that is crucial only for *GEOS*.
- Our best implementation can compute the complete spatial self join of all the 1.4 billion geometric objects from the complete OpenStreetMap data (with a total size of 31.1 billion triples) in less than 3 hours on a commodity PC. This was out of each reach for any other existing implementation we tried. In particular, PostgreSQL+PostGIS takes weeks for this computation.

All our code and materials are open source and available on GitHub, including documentation and instructions for how to reproduce the results from this paper. The repository can be found at https://github.com/ad-freiburg/spatialjoin.

2 Related Work

The typical way to perform a spatial join, which we also use, consists of the following three phases: (1) retrieve all pairs of geometries, called *candidates*, where the bounding boxes intersect; (2) optionally reduce the set of candidate pairs further, using heuristics; (3) for each candidate pair, compute the exact geometric relations. We will discuss the related work for each of these phases.

2.1 Candidate Retrieval

A good overview on algorithms for candidate generation can be found in the survey [14]. It focuses on intersection, whereas we consider the full DE-9IM matrix. The survey distinguishes between internal-memory methods (like R-tree indexing and other hierarchical data structures, also see [22], [19], [28]) and external-memory methods (like sweep line techniques). The basic idea of sweep line techniques is to sweep a vertical line over all bounding boxes, in order of ascending x-coordinate, and to maintain the set of rectangles that intersect the current sweep line. The typical data structure for that set is an interval tree. This technique can be used to report all intersecting pairs in a set of axis-aligned rectangles in $O(n \log n)$ [23, 25]. Our approach (see Section 3.1) builds on this technique.

2.2 Candidate Set Reduction

Brinkhoff and Kriegel [6] propose a set of geometric approximations to reduce the candidate set, like the rotated minimum bounding rectangle, the minimum bounding circle, or the convex hull. Another way of approximating geometries is by overlaying a grid over the geometric space and representing a geometry by its intersecting grid cells. Azevedo et al. [3] present a three-color covering, which for each geometry stores the set of cells which it fully covers and the set of cells which it partially covers. We describe and analyze a similar approach in Section 3.7. Georgiadis and Mamoulis [12] use a four-color scheme with the additional information whether a cell is covered more or less than 50%. They additionally enumerate the cells using the Hilbert curve, which allows for efficient compression and intersection. Google's *S2* library[2] combines the Hilbert-curve approach with a hierarchical grid that can be stored efficiently. S2 does not use colors, but explicitly maintains an outer covering (a set of cells which fully covers the geometry) as well as an inner covering (a set of cells which is fully covered by the geometry).

2.3 Full Geometry Comparisons

For full geometry comparisons, we again rely on a sweep line algorithm. Bentley and Ottmann extend the algorithm by Shamos and Hoey to report *all* intersections in a set of line segments [5]. In this scenario, the algorithm cannot stop on the first intersection, and line segments may change their order relative to the sweep line at crossing points. To handle this, intersection points are added as future events, which requires a dynamic event list. This is typically realized via a priority queue, which results in a running time of $O((n+k)\cdot\log n)$, where k is the number of reported intersections. An important special case of this problem is the red/blue line-segment intersection problem, where two sets A and B of line segments are given, with the property that no two lines from A and no two lines from B have intersecting interiors. Then the relative order of the

[2] https://s2geometry.io

active segments in A and in B remains fixed during the line sweep. Mairson and Stolfi [18] proposed an $O(n \cdot \log n + k)$ algorithm for this problem, which was later simplified by Chan [7]. We use an approach similar to Chan's in Section 3.3.1.

2.4 Related Problems

Several works solve variations of our problem by restricting the size of the input, the type of geometries, or the set of supported spatial predicates. Kipf et al. [16] discuss the problem where a relatively small and static set of polygons (e.g., the streets and buildings of a city) are joined with a large set of points that are streamed into the algorithm (e.g. the current position of drivers and potential passengers in a taxi app). They precompute an efficiently compressed quadtree-based index which stores a hierarchical, cell-based approximation of the polygons using Google's S2 library. In particular, they discuss applications where such an approximation is sufficient and the exact predicate evaluation can be omitted. Alhammadi et al. [2] first simplify all input geometries using a variation of the Douglas-Peucker algorithm and then run a standard filter and refine approach like described above. This leads to great speedups, but leads to a loss of precision (which is acceptable in many cases). Note that our approach also uses simplified geometries (see Section 3.6), but only in the filtering steps without compromising the precision of the result. Osborn [20] discusses the spatial join of two data streams where only a small fraction of both inputs can be stored at the same time. Aghajarian et al. [1], Geng et al. [11], and Zhang et al. [32] use GPUs to solve spatial joins efficiently. These approaches require that the complete input fits into the memory of the GPU. You et al. [31] discuss the problem of polyline intersection on GPU clusters. There also is much research on distributed spatial joins using frameworks like Apache Spark or Apache Hadoop (e.g. [17], [29], [10], [15], [30]).

2.5 Tools and Software

PostGIS[3] is a widely used extension for the *PostgreSQL* database that can compute spatial joins with the full DE-9IM matrix, like we do. It supports the precomputation of R-tree indices for columns with spatial data. When performing a spatial join, PostGIS only uses these indices for one of the inputs, even if indices for both inputs are present or a self-join is performed. For the exact predicate evaluations, PostGIS uses *GEOS*[4], a widely used library for geometric primitives. In Section 4, we evaluate the use of *GEOS*.

HyPerSpace [21] is the geospatial extension of the in-memory database *HyPer*; unfortunately, the code is proprietary. HyPerSpace relies on Google's S2 for the efficient processing of geospatial queries on datasets that change frequently.

OSCAR [4] allows specific spatial queries on the complete OSM data, namely finding all OSM objects in a given set of OSM regions. The core idea is to precompute a *cell arrangement* of all the given polygons. Then the mentioned spatial queries reduce to computing the intersection of lists of cell IDs. This approach can be considered as a variant of the grid approaches discussed in Section 2.2, making use of special properties of the OSM data (namely that the produced cells are relatively few and of similar size).

[3]https://postgis.net
[4]https://libgeos.org

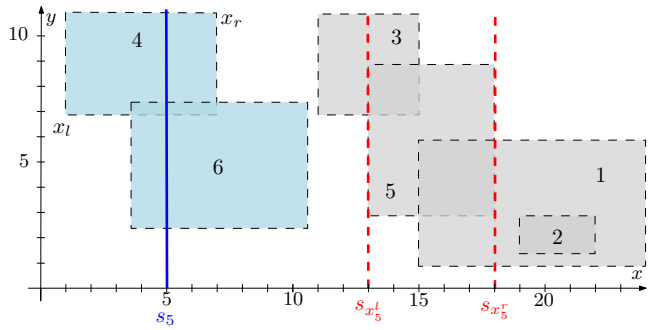

Figure 1: Reporting pairs of intersecting rectangles using a sweep line approach. For two rectangles to intersect, a vertical line must intersect both of them, We sweep such a line from left to right, checking each position. The set of rectangles intersecting s only changes at the left or right x coordinates (x_l and x_r) of rectangles.

3 Experimental Approach

Our basic approach consists of the following two steps: (1) candidate retrieval, which reports all geometries with intersecting bounding boxes, and (2) full geometry comparisons between candidates. We will first describe this baseline method, and gradually extend it with varying heuristics. The goal of all our heuristics is to keep the number of expensive full geometric comparisons low. The effect of our techniques will then be evaluated in Section 4.

3.1 Candidate Retrieval

For the candidate retrieval, we are given a set of axis-aligned bounding rectangles $B(G) = ((x_l, y_l), (x_r, y_r))$ for each geometry G. Our goal is then to find all pairs of intersecting rectangles. As described above, we use a sweep line approach to produce the candidate pairs.

Two axis-aligned rectangles $B(G_1)$ and $B(G_2)$ intersect if and only if there exists a vertical line s_x that intersects both. We may thus sweep such a vertical line over the dataset and retrieve at all x positions the rectangles intersecting s_x. For each of these *active* rectangles, we then check whether their y-intervals also overlap. The active set changes only at the x_l and x_r positions. Thus, it suffices to sweep along these x values (Figure 1).

All x_l and x_r and their corresponding object ID are stored as tuples (x_l, i, IN) and (x_r, i, OUT) in an *event* list E. Left x coordinates are an IN event, right x coordinates are an OUT event. If we now sort E by these x coordinates, an iteration over E is equivalent to sweeping s over the entire dataset, but skipping positions where the state does not change. On IN events, we add the rectangle ID to an *active set* A. On OUT events, we remove the ID from A.

The check for overlapping y intervals can be done by using an *interval tree* for the active set. The *insert*, *delete*, and *lookup* operations on thus interval tree can be done in $O(\log n)$. Finding overlapping y-intervals for a single rectangle takes time $O(\log n+k)$, where k is the number of overlapping intervals. This approach thus runs in $O(|O| \cdot (\log I + M))$, where O is the set of all geometric object, M is the maximum number of rectangles intersecting a single rectangle, and I is the maximum size of A at any time.

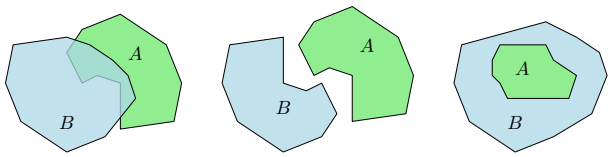

Figure 2: Two polygons have either intersecting boundaries, are disjoint, or one is completely contained in the other.

Note that E can be easily maintained, sorted, and traversed on disk. Thus, memory is only needed for the active set and the interval tree. For real-world input data, I is usually small (in our evaluation dataset OpenStreetMap, it was around 50,000).

3.2 Geometry Cache

So far, the candidate retrieval only delivered geometry IDs. For further checks, we need the actual geometric objects. As storing all full geometric objects in memory is unrealistic, we store them on disk and load them on demand. To avoid excessive disk access, we use a straightforward LRU cache. This has the effect that very large polygons requiring many comparisons (e.g. polygons of countries) usually remain in the cache until they are no longer required.

To further reduce both the I/O and the required disk space, we distinguish 5 different geometric types: **Points** are stored as raw 64 bit integer coordinates (x and y are 32 bits, respectively). **Simple Lines** have only 2 anchor points, stored as raw 64 bit integer coordinates. **Lines** that aren't *simple* are stored as lists of segments (possibly sorted, see Section 3.3.2), together with their length, their bounding box, and additional precomputed information used by our heuristics (see Sections 3.5 - 3.7 for details). **Simple Areas** are areas with less than 10 anchor points and without any holes, these are stored as raw lists of their 64 bit integer coordinates. **Areas** that aren't *simple* are stored as pre-sorted lists of segments, together with their holes (also possibly pre-sorted, see Section 3.3.2), their bounding box, their geometric area, and again additional precomputed information used by our heuristics. The idea of the distinction into "simple" and "normal" geometries is that for the simple geometries, all precomputations are cheaper to do on the fly than loading them from disk. For our experiments, we used one cache per geometry type, with a maximum size of 10,000 objects. We found that this limit provided a good trade-off between memory requirements and disk I/O.

3.3 Full Geometry Comparisons

The computation of the DE-9IM matrix is trivial for point/point pairs, as there is only one case: the points are equal (`0FFFFFFF2`). For the remaining pairs (point/line, point/polygon, line/line, line/polygon, polygon/polygon), we reduce all comparisons to two basic operations: (1) given two sets L_1 and L_2 of line segments, retrieve the set of intersecting line segments. (2) given a point P and a set L of line segments, find out whether P lies on a line segment of L (and which), or inside a closed polygon described by L.

For point/line comparisons, there are only two distinct cases: either the point is on the line (Operation 2), or the point is on one of the end points of the line (Operation 2, with subsequent check whether the found line segment is the first or last of the line, and whether the point is equal to its beginning or end). If none of these cases apply, then the geometries are disjoint. For point/polygon comparisons, again there are only two distinct cases: either the point is on a segment of the boundary (Operation 2), or the point is inside the polygon (also Operation 2). Again, if none of these cases apply, the geometries are disjoint.

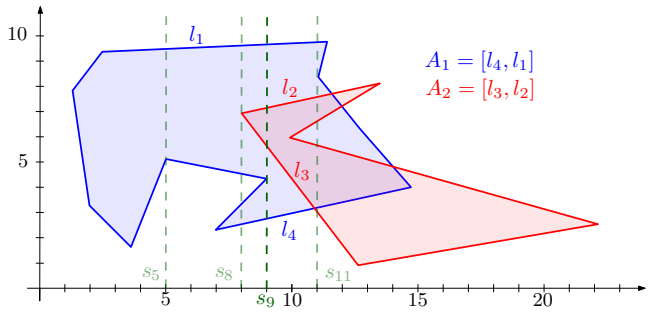

Figure 3: Sweeping over two sets of line segments (blue and red) to find intersections between blue/red and red/blue pairs. In this case, the line segments are polygon boundaries. A_1 and A_2 are the active sets of the two polygons at s_9.

For line/line comparisons, we use Operation 1 to find the intersecting line segments. We can then fill the DE-9IM matrix by analyzing only the intersecting line segments (see Section 3.3.3 for details). If no line segments intersect, the lines are disjoint.

For line/polygon comparisons, there are three distinct cases: (1) Operation 1 results in intersecting segments pairs. Then we can fill the matrix by analyzing the individual intersecting segment pairs (note that we can in particular find out whether any line segment crosses into the interior of the polygon as we know at each time which side of polygon's border segment is "inside", and which is "outside" the polygon). (2) Operation 1 results in no intersecting segment pair. Then the geometries are either disjoint, or the line is completely contained in the polygon. In both cases, the DE-9IM matrix is trivial to fill, and we can safely distinguish the two cases by a single Operation 2 for *any* point on the line.

The same approach can be used to compare two polygons (see Figure 2 for an example).

Operation 2 is the standard point-in-polygon tests, which can be solved in linear time by a ray casting algorithm [26]. We will describe Operation 1 in more detail below.

Note that we ignore polygons with inner rings (holes) here for brevity, although we considered them in our experiments. In principle, if a polygon with a hole is involved in a comparison, the inner rings have to be compared separately to the other side, using the same approach as described here. To avoid iterating over all inner rings of polygons for each check (which might be problematic for polygons with a large number of such inner rings), we store for each polygon the bounding boxes of each inner ring, together with a list of inner ring IDs sorted be the leftmost x-coordinate of the ring. This enables us to quickly search for the first relevant inner ring, and to discard them as irrelevant based on the bounding box.

3.3.1 Sweep Line Approach. The naive approach for Operation 1 is a pairwise intersection test between all pairs $(l_1, l_2) \in L_1 \times L_2$, resulting in time $O(|L_1||L_2|)$. As mentioned above, a restricted variant of Operation 1 is known as the red/blue segment intersection

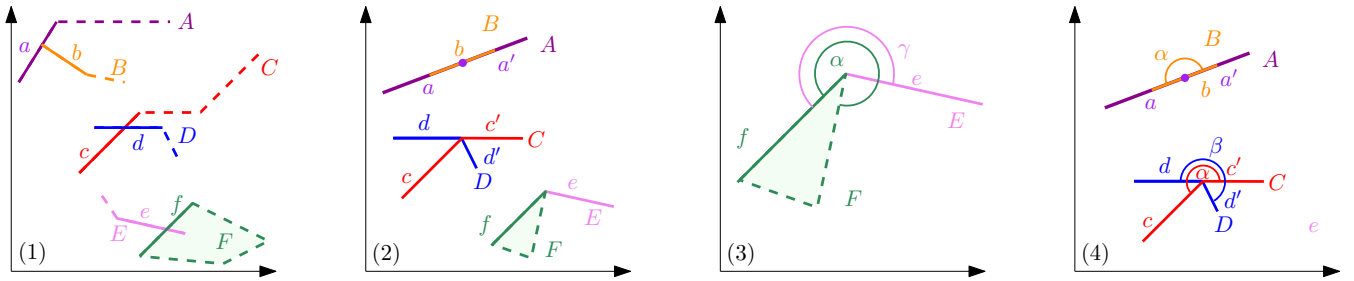

Figure 4: Using locally intersecting line segments to decide geometric relationships. (1): A local intersection analysis is enough to decide global relationships. For example, segments a and b touch at an end point of b (which is a boundary point of B), and A and B thus have a DE-9IM matrix of `F01FF0102`. (2): Cases where local analysis is not enough: segment b both locally overlaps segments a and a', but line A covers line B. For C and D, we would locally decide that C and D touch, although they cross. e is to the right of f, but does not lie inside F, as another segment restricts the interior. (3) For each segment, we store the outgoing angles of all adjacent segments. Then we can locally decide that e does indeed not cross into F by comparing α to γ. (4) Similarly, we can now decide that based on the comparison a vs. b, b is not contained in A's exterior, as the outgoing angle of the adjacent segment is exactly $180°$. For C and D, the presence of an outgoing angle at the intersections of d and c makes it clear that the local touch of d and c is not a global touch.

problem: In this variant, the interiors of all line segments in L_1 (and L_2 respectively) are only allowed to intersect in their endpoints (that is, interior intersections are only allowed for pairs $(l_1, l_2) \in L_1 \times L_2$). This restriction can be easily fulfilled in our case: we simply pre-process all input geometries and add explicit anchor points at places where segments intersect. The red/blue segment intersection problem can be efficiently solved using a sweep line algorithm [7, 18].

We first insert all line segments from L_1 and L_2 into a merged set L and mark each line segment as originating either from L_1 or L_2. We assume that for a line segment $l = ((x_1, y_1), (x_2, y_2))$, it already holds that $x_1 \leq x_2$ and use the more convenient notation $((x_l, y_l), (x_r, y_r))$. Next, we again build an event list E, into which each line segment l is inserted as two events: (x_l, o, l, IN) and (x_r, o, l, OUT), where $o \in 1, 2$ states whether l is from L_1, or L_2. E is again sorted by the left and right x coordinates.

Again we require an active set that at each sweep line position s_x holds segments intersecting s_x, sorted by the y-coordinate of their intersection with s_x. For two line segments l and k, we say $l <_x k$ if the y-coordinate of the intersection of s_x with l is smaller than that of the intersection of s_x with k (we break ties in ambiguous cases, for example if an endpoint of l is exactly on k). We now exploit the fact that no two segments from L_1, and no two segments from L_2, have intersecting interiors. This means that during sweeping, the relation $<_x$ is *constant* for each pair $(l_1, l_1') \in L_1^2$, and each pair $(l_2, l_2') \in L_2^2$: no two line segments from the same set can ever "switch sides". We can thus maintain two *separate* active sets A_1 and A_2, and keep them correctly sorted according to $<_x$ using the following simple ordering relation: if a segment m was added to A before a segment n (if $x_{ml} < x_{nl}$), then $n < m$ if x_{nl} is to the right of m. If $x_{ml} > x_{nl}$, then $n < m$ if x_{ml} is to the left of n. If the left endpoint of n is on m (or vice versa), we use the right endpoint. If the left endpoint of n coincides with the *right* endpoint of m, we break ties by always ordering $n < m$. In our experiments, we use a simple binary search tree (the C++ STL's `std::set`) for A_1 and A_2. Figure 3 gives an example of such a line sweep.

Going over the event list E, we then have to consider the following cases:

(1) On event (x_l, o, l, IN): add x_l to A_o and search the other active set $A_{\bar{o}}$ for the line segments directly above and below of x_l (if they exist). In both directions, check whether x_l intersects with them and report if they do (e.g. at s_8 in Figure 3). If there is an intersection in one direction (as is the case at s_{11} in Figure 3), continue traversing and checking $A_{\bar{o}}$ in this direction until the first non-intersecting line segment is encountered. Note that the total number of these checks is bounded by $O(k)$, where k is the maximum number of intersections.

(2) On (x_l, o, l, OUT): remove x_l from A_o. Note that x_l might have been a "blocker" masking an intersection between a line segment from $A_{\bar{o}}$ directly below x_l and a line segment from A_o directly above x_l (or vice versa). This is for example the case at s_9 in Figure 3. We check all lines below x_l in $A_{\bar{o}}$ and all lines above x_l in A_o for pairwise intersections (and vice versa), again stopping in either direction as soon as we cannot find any more intersections. Again, the total number of these checks is bounded by $O(k)$.

Sorting the event lists takes time $O(|L| \log |L|)$, and sweeping over the events takes time $O(|L| \log \max(|L_1|, |L_2|) + k)$, resulting in a total running time of $O(|L| \log |L| + k)$. Note that $k \in O(|L_1| \cdot |L_2|)$.

3.3.2 Pre-Sorted Geometries. A key benefit of this approach is that we can store and pre-sort the event lists E_1 and E_2 for L_1 and L_2. For reporting the intersections between L_1 and L_2, we then do not have to materialize and sort the combined event list E: it is enough to iterate over the pre-sorted event lists E_1 and E_2 in a "zipper"-like fashion, producing the sorted event list E on the fly.

It is important to understand why this is relevant for our practical performance: theoretically, the difference between the bare sweep complexity $O(|L| \log \max(|L_1|, |L_2|) + k)$ and the total (including sorting) complexity $O(|L| \log |L| + k)$ does not appear to be very significant. However, the $O(\log \max(|L_1|, |L_2|))$ part comes from the

lookup operations in the binary search trees holding the active sets. In practice, these sets are extremely small - for a convex polygon, they contain at most 2 segments, and in our testing datasets, the active set was typically smaller than 10. Also note that we have to do $k = |L_1| \cdot |L_2|$ additional iterations of the binary search tree in up and down direction only in the worst case - typically, k is very small. For almost all comparisons in our testing datasets, the sweeping can thus be assumed to run in linear time (and it strictly runs in linear time for comparisons between convex geometries), while the sorting of the merged event list E would have still required $O(|L| \log |L|)$ time. Additionally, we save the copying of E_1 and E_2 into a sorted merged list E. This is highly relevant in our scenario because we typically have millions of comparisons involving the same geometry - for example, in our testing dataset of the entire German OpenStreetMap data, we require (without any heuristic) a geometric check of all $\sim 84M$ houses in Germany against the geometry of Germany itself. Without the pre-sorting, we would have to sort the line segments constituting the German border 84 million times.

3.3.3 Analyzing Line Segment Intersections. If any two line segments l_1 and l_2 intersect, we would like to know the following: (1) Do their geometries only touch at exterior bounds, or do they cross each other? (2) Do they overlap? For checks against a part of a polygon boundary l_2, we would additionally like to know whether (3) l_1 lies on the inside or the outside of the polygon. This would enable us to fill the DE-9IM matrix between the corresponding geometric objects.

Checking this seems straightforward: (1) and (2) are easy to check, and (3) is a matter of storing the polygon boundary l in its original clockwise orientation, in which case any intersecting line segment crossing to the right of l is on the "inside" side of the segment, and any line segment crossing to the left on the "outside" side. Figure 4.1 gives examples. However, consider Figure 4.2. Here, we would for example decide that e crosses into the interior of the polygon corresponding to f - but this is not actually the case, as a neighboring line segment further restricts the interior of the polygon. A similar problem can be seen between C and D: here, we would locally decide that two lines touch each other, while in reality they cross each other. To handle these problems, we additionally store for each line segment the left and right outgoing angle of the adjacent line segment as a *lookahead*. If the segment was the last (or first) segment of a line (that is, its boundary), we use a special placeholder value indicating "no adjacent segment". We can then avoid false local reporting by comparing angles, as shown in Figures 4.3 and 4.4. Note that the angles are stored as floating point numbers and are compared without any tolerance.

3.4 Skipping Irrelevant Segments

The pre-sorting of line segments by their left x-coordinate enables other speed up heuristics. Consider Figure 5. It is easy to see that all line segments of B that end to the left of A are completely irrelevant for the sweepline process - they will have left the active set of B before any segment of A will become active. We may thus skip them completely by iterating to the first segment which is actually relevant for our process. If there are n segments to the left of A, this still requires iterating over n segments. As the list of

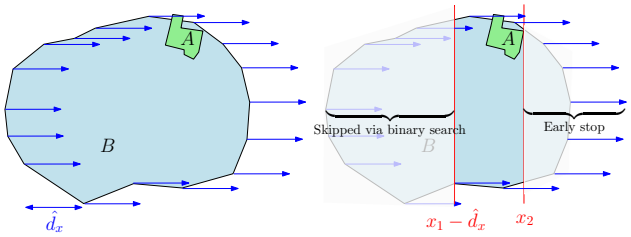

Figure 5: Comparing a large geometric B object to a smaller object A. In the list of sorted segments of B, we can skip all segments with $x_l < x_1 - \hat{d}_x$ using a simple binary search. When we reach a segment of B with $x_l > x_2$, we abort.

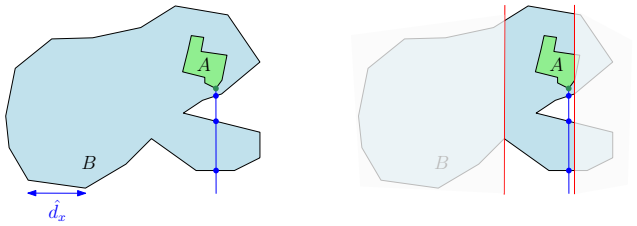

Figure 6: Checking whether a single random point of A is in B using raycasting with a ray $(x, -\infty)$ to (x, y). We can again search for the first segment of B with with $x_l >= x - \hat{d}_x$

segments is sorted, we would like to apply a simple binary search to get the number of iterations down to $O(\log n)$. But we cannot simply search for the first segment of B which ends after the first segment of A: the segments are sorted by the *left* x-coordinate. To nevertheless apply a binary search, we store for each geometric object the *longest* covered x-interval $\hat{d}_x$. Let x_a be the leftmost x-coordinate of A. Using binary search, we can then quickly search for the first segment of B with $x_l >= x_a - \hat{d}_{Bx}$. We can also trivially abort our sweep line process as soon as we are past the rightmost segment of A, effectively restricting our sweep line process to a narrow band around A.

Note that the same technique can be used to speed up the point-in-polygon tests (Operation 2). Consider Figure 6. The standard ray-casting algorithm iterates over all line segments and checks for an intersection between a line going from infinity to $P = (x, y)$. If we chose this line to go from $(x, -\infty)$ to (x, y), we again can use a binary search to find the first segment of B with $x_l >= x - \hat{d}_{Bx}$. If we then check the line segments for intersections with $((x, -\infty), (x, y))$ in their sorted order, we can again abort as soon as the first segment of B with $x_l > x$ appears.

3.5 Surface Area Precomputation

Recall that Operation 2 (Point-in-Polygon check) is required to differentiate between two cases after we have established that the segments of a geometry A and a polygon B do not intersect: (1) A is completely contained in B, or (2) A and B are disjoint. If both A and B are polygons, we have to both check if A is in B, and (if not), if B is in A. A simple preprocessing step to avoid one of these checks is to compute the surface area of all input polygons. Then, if the surface area of polygon A is smaller than the surface area of polygon B, we can already be certain that A cannot contain B.

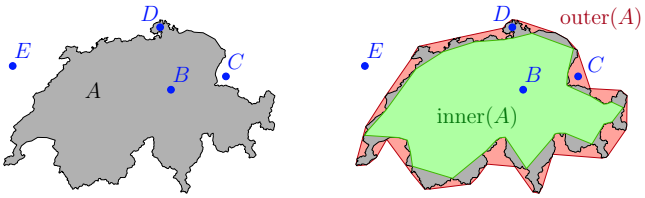

Figure 7: Using inner and outer simplified geometries for faster geometry comparison. If B is contained in the inner geometry, it is surely contained in A. If E is not contained in the outer geometry, it is surely disjoint with A.

3.6 Approximate Geometries

The techniques described so far aim to improve the performance of raw geometry comparisons. The reminder of this section will discuss heuristics to avoid such comparisons between full input geometries. Our techniques can be broadly classified into two categories: (1) *approximate geometries*, which are then again geometrically compared, and (2) a decomposition of input geometries using a static cell grid, which can then be used to decide some geometric relations without any geometric comparisons. We will first give a list of the simplified geometries used in our evaluation.

3.6.1 Diagonal Bounding Box. A bounding box simplification is already used in the geospatial index. This may be further refined by also computing *diagonal* bounding boxes - that is, the axis-aligned bounding box after the coordinate system has been rotated by 45°. The fixed orientation enables us to check for intersections using standard intersection tests for axis-aligned rectangles. Additionally, the boxes can be stored using only 2 coordinates. We store them directly in the event list for the candidate retrieval described in Section 3.1 (all other precomputations are stored in the cache). This heuristic may thus completely bypass the geometry cache.

3.6.2 Oriented Bounding Box. To improve on the diagonal bounding box, while still keeping the number of anchor points at only 4, we additionally precompute the oriented bounding box (OBB) [27] of each polygon in O and use this OBB to quickly decide whether two geometries are disjoint.

3.6.3 Inner and Outer Ramer-Douglas-Peucker. For better approximation, and to enable fast positive containment or intersection decisions based on simplified geometries, we also precompute for each polygon A two approximations: a simplified *outer* polygon $\text{outer}(A)$, and a simplified *inner* polygon $\text{inner}(A)$. It is clear that $B \subseteq \text{inner}(A) \Rightarrow B \subset A$ and $B \not\subset \text{outer}(A) \Rightarrow B \not\subset A$, and also $G \cap \text{inner}(A) \neq \emptyset \Rightarrow G \cap A \neq \emptyset$ and $G \cap \text{outer}(A) = \emptyset \Rightarrow G \cap A = \emptyset$.

To compute $\text{outer}(A)$ and $\text{inner}(B)$, we modify the classic Ramer-Douglas-Peucker (RDP) algorithm for line simplification [9, 24]. Given a line L as an ordered list of anchor points, RDP takes an anchor point pair (p, u) (starting with the first and last point of L) and finds the point q between them with the largest distance $d = \text{dist}(q, p, u)$ to the line segment $\overline{pu}$. If d is smaller than a simplification threshold ϵ, all points between p and u are discarded. If $d > \epsilon$, q is kept, and the process recursively continues for (p, q) and (q, u).

Given a polygon A as a closed list of anchor points $(p_1, \ldots, p_n)$, we simplify the lines $(p_1, \ldots, p_{\lfloor n/2 \rfloor})$ and $(p_{\lfloor n/2 \rfloor + 1}, \ldots, p_n)$ separately and later join the resulting simplified lines again. As a simplification criterion for RDP, we use the signed distance function

$$\text{dist}(q, p, u) = \frac{(x_u - x_p)(y_p - y_q) - (x_p - x_q)(y_u - y_p)}{\sqrt{(x_u - x_p)^2 + (y_u - y_p)^2}} \quad (1)$$

for a point $q = (x_q, y_q)$ and a line segment described by $p = (x_p, y_p)$ and $q = (x_q, y_q)$.

For the inner polygon, we keep q if $0 < \text{dist}(q, p, u) < \epsilon$. For the outer polygon, we keep q if $0 < -\text{dist}(q, p, u) < \epsilon$.

Note that for non-convex polygons, discarding a point to the right of $\overline{pu}$ does not necessarily mean that the resulting line segment is inside the original polygon. Similarly, discarding a point to the left does not mean that the resulting line segment is outside of the polygon. A simple mitigation strategy is to check a posteriori whether $inner(A)$ is really contained in A, and to give up computing $inner(A)$ otherwise. Similarly, if A is not contained in $outer(A)$, give up computing $outer(A)$.

During our experiments, we found that selecting a fixed ϵ for the inner and outer simplified polygons is nontrivial. Small ϵ values lead to little to no gains for large polygons. For smaller polygons, a large ϵ will usually result in empty inner geometries, and an outer geometry that is equivalent to the convex hull. We settled on the following dynamic parameter: $\epsilon(A) = \alpha\sqrt{\text{area}(A)/\pi}$. This bases $\epsilon(A)$ directly on the (weighted) radius of a hypothetical circle with the same surface area as A, so that larger geometries are simplified more. We set the weight $\alpha = \frac{1}{20}$.

3.7 Intersecting Cell IDs

So far, the speed-up techniques still relied on geometric comparisons. This section describes a technique which allows to quickly decide whether a geometry G is definitely contained in a polygon P, whether it is definitely *not* contained in P, or whether it *might* be contained, without doing any geometric calculation (apart from the preprocessing). This technique is based on a grid covering the dataset. The grid cells are continuously numbered from east to west, and north to south. We have found this approach to be more efficient than the more sophisticated hierarchical cell-covering from Google's *S2* library (see section 2.5), mostly because of the more expensive precomputation of the covering in S2.

3.7.1 Collecting Cell IDs. In a preprocessing step, we first compute for each polygon P the set A^+ of grid cells completely covered by P and the set A^- of grid cells only intersecting P. We define $A = A^+ \cup A^-$. The cells are represented by their integer id. Second, we compute for each line L the set A^- which holds all cells intersecting L (note that A^+ is naturally empty for lines). Note that A^- for points can easily be determined on the fly (it contains a single cell ID).

3.7.2 Geometric Contains and Intersect via Efficient List Intersection. Consider two polygons P_1 and P_2 and their cell ID sets A_1^+, A_1^-, A_2^+, and A_2^-. If $|A_1 \cap A_2^+| = |A_1|$, then P_1 is surely completely contained in P_2, and the DE-9IM matrix is trivially `2FF1FF212`. If $|A_1 \cap A_2| = 0$, then P_1 and P_2 are surely disjoint. In all other cases, a full geometry check is required. Similar decisions can be made for line/line pairs, line/polygon, point/line and point/polygon pairs.

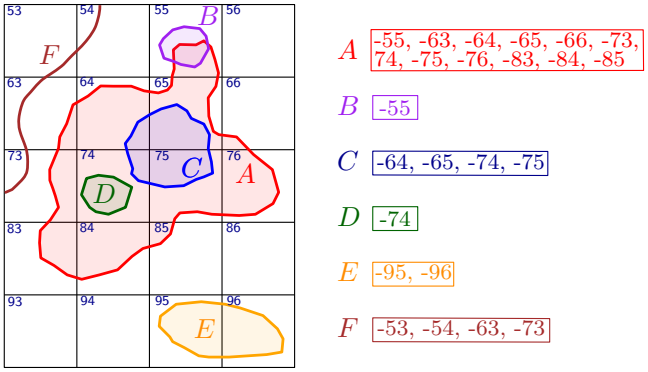

Figure 8: A static grid covering the bounding box of the entire dataset, with numbered cells. If a geometric object intersects a cell, it is added with a negative sign to its list of cell IDs. If a cell is completely contained in a polygon, the cell ID is added with a positive sign.

To compute these measures in a single pass, we combine A^+ and A^- in a list $L = (c_1, \ldots, c_n)$ in which cell IDs from A^+ are stored unchanged, and cell IDs from A^- are stored with a negative sign. L is then sorted by $|c_i|$. Computing both $|A_1 \cap A_2|$ and $|A_1 \cap A_2^+|$ is then a list intersection problem between L_1 and L_2, sorted by the (absolute!) values of their cell IDs. We compute this with an exponential search approach.

3.7.3 Efficient Calculation and Storage of Cell ID Lists. A naive computation of the cell ID sets (A^+, A^-) for a polygon P would first collect all cells c intersecting P's bounding box, then determine whether each c is fully contained in P or only intersects it. This would require a number of box-in-polygon checks which depends on the surface area of P, and might thus be quadratic in the worst case. To mitigate this, we scan P in a quadtree-like fashion. Let w be the side length of a grid cell, and let W and H be the width and height of the bounding box of p. We call grid cells of side length w a *base cell*. For our experiments, we set w to 1 km. Instead of scanning P with many small cells of width and height w, we begin with cells of width $\lceil W/w \rceil / 4 \cdot w$ and height $\lceil H/w \rceil / 4 \cdot w$. These larger *probe* cells are grid-aligned. Each of them contains $\lceil W/w \rceil / 4 \cdot \lceil H/w \rceil / 4$ base cells. Let d be such a cell. Then, if d is fully contained in P, we add all contained base cells to A^+ and proceed to the next cell. If d does not intersect with P, we skip it completely. If d intersects with P, but is not contained, we subdivide it into 4 smaller cells and check P against these cells. This process continues recursively. If d is a base cell, the recursion stops, and we add the corresponding base cell to A^-.

To avoid excessive memory consumption of the cell ID lists, we additionally use a simple run-length encoding. Note that if a fully contained *probe* cell contains $n \cdot m$ base cells, it is then sufficient to simply add the cell with the lowest x value and a running length of n for each of the m rows.

4 Experimental Evaluation

We have implemented the approach described in Section 3 as a library and command-line tool. Our tool inputs two sets of geometries O_1 and O_2 and outputs a set of triples with all the geometric relations, as defined in Section 1.1. Specifically, the input is a single TSV file with one line per geometry and three columns: a unique ID for the geometry, an index indicating to which of the two input sets the geometry belongs (0 for O_1 and 1 for O_2), and a WKT representation of the geometry. The output is one line per intersecting geometry pair, in the format `<id1> <DE-9IM> <id2>`. The code is publicly available on https://github.com/ad-freiburg/spatialjoin.

Our evaluation focuses on two main aspects. First, the effect of using different implementations for the full geometry comparisons, namely our own implementation based on pre-sorted geometries as described in Section 3.3.1 (**Sorted**), versus the implementation from the *GEOS* library based on a conventional representation (**Geos**). Second, the effect of our five heuristics on **Sorted** and **Geos**.

Table 1: The number of the different kinds of geometries in each of our four datasets, the number of candidate pairs for the self join, and the total number of result pairs.

	points	lines	poly.	coll.	multi.	#cands	#res
FIN	1.6 M	3.5 M	4.6 M	19.5 K	6.2 K	226 M	178 M
GER	20.0 M	19.7 M	47.9 M	0.5 M	17.8 K	3.1 B	1.5 B
OHM	5.4 M	2.4 M	1.4 M	17.8 K	7.9 K	1.2 B	196 M
OSM	264 M	327 M	777 M	5 M	0.4 M	37.8 B	31.1 B

4.1 Setup

For each evaluation, we first parse the input WKT strings into either our pre-sorted representation or the *GEOS* representation. The code for converting the WKT strings into lists of coordinates is the same for both representations. Afterwards, any optional pre-processing required for the heuristic is done, and the geometry (and all pre-computed information) is stored in the geometry cache described in Section 3.2 We then use the sweep line approach described in Section 3.1 to generate a stream of *candidate pairs*, that is, pairs of geometries with intersecting bounding boxes. For each of these pairs, we then load the geometry from the geometry cache and try to infer the DE-9IM matrix using any of the enabled heuristics. If the DE-9IM matrix can not be determined by one of these heuristics, we compute the exact geometric relations using either the pre-sorted geometries as described in Section 3.3.1 (**Sorted**), or using the `GEOSRelate` method provided by the *GEOS* library (**Geos**). Note that apart from this last step, the setup is identical for **Sorted** and **Geos**.

All our experiments were run on a commodity PC, with an AMD Ryzen 9 9950X processor with 16 physical and 32 virtual cores, 186 GB of RAM (DDR5), and 29 TB of disk space (NVMe SSD), running Ubuntu 24.04. For each evaluation, we used 28 cores.

4.2 Evaluation on Full Self Joins

We first evaluate on full self joins, the most expensive kind of spatial joins. In a spatial self join, the two input sets from our definition in Section 1.1 are the same.

We consider four datasets: all geometries from OpenStreetMap (OSM), the subset that lies in Germany (GER), the subset that lies in Finland (FIN), and all geometries from OpenHistoricalMap (OHM). The dimensions of these datasets are shown in Table 1. FIN and

Table 2: Effects of our heuristics if the exact geometry relations are computed using our pre-sorted geometries (Sorted). Each time is for computing the self join of the respective dataset for the respective setting of our heuristics, as described in Section 4.2. The speedup is relative to the setting shown in the last column (which is the same for each of the last three rows).

	FIN		GER		OHM		OSM		
heuristics	time	speedup	time	speedup	time	speedup	time	speedup	relative to
csdoi	0.8 min		14.8 min		88.9 min		1055 min		
Csdoi	0.7 min	1.26 ×	10.0 min	1.48 ×	28.0 min	3.17 ×	181 min	5.83 ×	csdoi
CSdoi	0.7 min	1.00 ×	9.9 min	1.01 ×	28.0 min	1.00 ×	188 min	0.96 ×	Csdoi
CSDoi	0.6 min	1.11 ×	8.7 min	1.15 ×	30.3 min	0.92 ×	165 min	1.14 ×	CSdoi
CSdOi	0.7 min	0.99 ×	10.0 min	0.99 ×	27.6 min	1.01 ×	186 min	1.01 ×	CSdoi
CSdoI	0.7 min	0.99 ×	9.8 min	1.01 ×	27.5 min	1.02 ×	176 min	1.07 ×	CSdoi

Table 3: Effects of our heuristics if the exact geometry relations are computed using the *GEOS* library (Geos). Each time is for computing the self join of the respective dataset for the respective setting of our heuristics, as described in Section 4.2. The speedup is relative to the setting shown in the last column (which is the same for each of the last three rows).

	FIN		GER		OHM		OSM		
heuristics	time	speedup	time	speedup	time	speedup	time	speedup	relative to
csdoi	465.9 min		> 10 h		> 10 h		>30 h		
Csdoi	97.4 min	4.79 ×	101.0 min	–	310.6 min	–	>30 h	–	csdoi
CSdoi	97.8 min	1.00 ×	103.7 min	0.97 ×	299.2 min	1.04 ×	>30 h	–	Csdoi
CSDoi	98.3 min	0.99 ×	99.0 min	1.05 ×	311.7 min	0.96 ×	>30 h	–	CSdoi
CSdOi	97.5 min	1.00 ×	100.2 min	1.03 ×	298.4 min	1.00 ×	>30 h	–	CSdoi
CSdoI	97.6 min	1.00 ×	63.9 min	1.62 ×	268.8 min	1.11 ×	>30 h	–	CSdoi

OHM are of roughly the same size, but have very different characteristics: OHM has many similar large regions, which makes self-joins significantly harder (because no heuristic we know of can filter out a pair of very similar objects).

For each dataset, we evaluated the effect of each of our six heuristics from Section 3:

- **C** cell IDs (Section 3.7)
- **S** precomputed surface area (Section 3.5)
- **D** diagonal bounding box (Section 3.6.1)
- **O** oriented bounding box (Section 3.6.2)
- **I** inner/outer simplified geometries (Section 3.6.3)

Each of these six heuristics can be switched on or off. We evaluated all of the resulting 64 combinations. We found that all significant effects can be observed from the relative speed-ups of these six combinations: csdoi, Csdoi, CSdoi, CSDoi, CSdOi, CSdoI. Each combination is represented by a six-letter string: an uppercase letter means that the heuristic is enabled, a lowercase letter means that the heuristic is disabled. The first combination is the baseline, with all heuristics disabled. The next two combinations successively add the heuristics C and S. The last three combinations add *either* of the three approximation heuristics D, O, or I from Section 3.6.

Tables 2 and 3 report the total running time of the self joins of each of the four datasets and for each of the six combinations. The only difference is that Table 2 shows the results when using the pre-sorted geometries described in Section 3.3.2 (**Sorted**), while Table 3 shows the results when using the conventional geometry representation provided by the *GEOS* library (**Geos**) instead. The tables provide five key insights.

First, computing the required full geometry comparisons based on the pre-sorted geometries (**Sorted**) vastly outperforms computing them using the conventional representation (**Geos**). For all datasets, when at least C is active, **Sorted** is more than 10 times faster than **Geos**. For FIN, the factor is over 100 because of Finland's various lakes with extremely complex geometries. These are no problem for **Sorted**, but very expensive when using **Geos**.

Second, C is the most effective heuristic for both **Sorted** and **Geos** and crucial for good performance. The larger the dataset or the more complex the geometries, the more important this heuristic becomes. In particular, C is very effective for settling pairs that consist of a large and complex geometry like a region boundary and a small and simple geometry like a building not too close to that boundary. With **Sorted**, exact comparisons are still feasible for such pairs because the binary search over the sorted line segments skips most of the boundary. For **Geos**, however, such comparisons are very expensive.

Third, I is important for **Geos** but not so much for **Sorted**. The reason is again pairs of a small and simple geometry, and a large and complex geometry, but this time for the case where the simple geometry is closer to the complex geometry. Such pairs can not be settled by C, but can be settled by I when the inner or outer geometry is a good enough approximation in the region relevant for the comparison. For FIN, there is no improvement, again because of the many complex lake boundaries (their approximate outer geometries are usually very close to a combination of their bounding box and their diagonal bounding box). For **Sorted**, the benefit is smaller for the same reason explained in the previous paragraph;

the additional cost of computing the inner-outer approximation is therefore typically not worth it.

Fourth, the diagonal bounding boxes `D` provide some improvement for **Sorted** for three of the four datasets. This is practically relevant because the diagonal bounding boxes are stored directly in the event list of the sweep and thus also avoid costly geometry cache loads for false-positive candidates. However, this backfires for **OHM** with its many large boundaries, where the cache is not evicted as often as it should be, which leads to a RAM consumption beyond the machine's swapping threshold. This is an artifact of our implementation, which can and should be fixed. For **Geos**, the effect of `D` vanishes in comparison to the much larger cost for the full geometry comparisons, and the performance drop for **OHM** is less pronounced for the same reason.

Fifth, the other heuristics have little effect for both **Sorted** and **Geos** and are therefore not relevant in practice. In particular, the oriented bounding box `O` is not worth the additional effort of computation and storage. And the very simple `S` has hardly any effect.

4.3 Evaluation on Other Spatial Joins

We also evaluated the following spatial joins, using the optimal heuristic for **Sorted** (CSDoi) and for **Geos** (CSdoI). The geometries are subsets of the same OSM dataset used in the evaluation above. Apart from the queries, the setup was the same as in the previous section. The results are shown in Table 4.

Q1 1 region (Germany) ⋈ 66.4 M residential streets
Q2 0.9 M powerlines ⋈ 66.4 M residential streets
Q3 66.4 M residential streets ⋈ 66.4 M residential streets
Q4 0.7 M administrative regions ⋈ 66.4 M residential streets

Q1 is a spatial join between a single large polygon and many small line segments. In that case, the `C` heuristic is able to quickly compute the result for most candidate pairs. The remaining pairs are few, so **Sorted** has only a slight advantage over **Geos** here. We note that most of the time for this query is spent parsing of the input geometries; the actual DE-9IM computation takes around two times longer for **Geos** than for **Sorted**.

Q2 mainly involves candidate pairs with simple and short line segments on one side of the spatial join, and simple but long line segments on the other side. For such comparisons, the geometry representation makes little difference. For both **Sorted** and **Geos**, the `C` heuristic helps to avoid many full comparisons.

Q3 mainly involves candidate pairs with simple and short line segments on both sides. In that case, the geometry representation makes even less of a difference. Also, the `C` heuristic has hardly any effect because the two geometries typically lie in the same cell.

Q4 is the hardest of the four queries. Is is comparable to Q1, except that now many complex geometries are involved on the left side of the spatial join. We therefore see the same effects as for Q1, but more pronounced.

In summary, the pre-sorted geometries make only little difference for these queries. The good news is that the additional effort (for pre-sorting the geometries) does not make the spatial join more expensive compared to using the conventional representation.

Table 4: Running times for the four spatial-join queries from Section 4.3. The second column gives the number of pairs for which the DE-9IM matrix was computed. The third column gives the number of candidates generated by the sweep line algorithm. The fourth column gives the number of pairs for which a full comparison was computed, using either Sorted or Geos.

				Full comparison using	
	#results	#cands	#exact	**Sorted**	**Geos**
Q1	2.0 M	2.5 M	10 k	1.1 min	1.2 min
Q2	1.4 M	73 M	12 M	1.2 min	1.2 min
Q3	84 M	100 M	100 M	1.2 min	1.4 min
Q4	298 M	417 M	34 M	2.7 min	6.4 min

4.4 Comparison Against Other Systems

Our evaluation above has compared variants of itself, regarding various heuristics and regarding the implementation of the full comparison of two geometries. We also ran our queries on other systems that compute spatial joins, notably PostgreSQL+PostGIS, and a variety of other well-known databases.

As explained at the beginning of Section 1, PostgreSQL+PostGIS uses a simple index nested-loop join for the candidate generation, and then uses the **Geos** library for the full geometry comparisons. The running times of PostgreSQL+PostGIS are therefore strictly worse than reported in Tables 3 and 4 for **Geos**. Vice versa, our **Sorted** approach would directly benefit PostgreSQL+PostGIS and other systems of its kind. And, of course, for datasets and queries like the above, these systems would benefit from using a sweep line approach for the candidate generation (with the exception of Q1, where the left side of the spatial join consists of only a single object).

5 Conclusions and Future Work

We provide a complete, fully functional, and carefully engineered implementation for the standard spatial-join problem. Our code as well as all the reproducibility materials are publicly available on https://github.com/ad-freiburg/spatialjoin. We investigate five heuristics for reducing the set of candidate pairs, as well as two variants for the exact geometry comparisons. The five key insights from our evaluation are: (1) our pre-sorted representation is crucial for spatial joins involving complex geometries, especially in combination with small geometries; (2) cell IDs are crucial for OSM-like data; (3) inner-outer bounding boxes are crucial when using **Geos**, while diagonal bounding boxes are better when using **Sorted**; (4) the other heuristics we tried have little practical relevance.

Interesting directions for future work are: (1) low-level optimizations like a special handling of convex geometries (which intersect the sweep line at most twice); (2) improve the handling of pairs of large but similar geometries, as they occur frequently in OHM; (3) re-run our experiments on synthetic datasets, e.g. generated by Spider[5]; (4) provide our implementation as a library that can be directly integrated into widely used systems like PostgreSQL+PostGIS.

[5] https://spider.cs.ucr.edu

References

[1] Danial Aghajarian, Satish Puri, and Sushil K. Prasad. 2016. GCMF: an efficient end-to-end spatial join system over large polygonal datasets on GPGPU platform. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2016, Burlingame, California, USA, October 31 - November 3, 2016*. ACM, 18:1–18:10. doi:10.1145/2996913.2996982

[2] Fatima Ahmed Alhammadi, Haya Almadhloum Alsuwaidi, Shooq Abdelrahman Alzarooni, and Isam Mashhour Al Jawarneh. 2024. Line Simplification for Efficient Approximate Join Queries On Big Geospatial Data. In *Fifth International Conference on Intelligent Data Science Technologies and Applications, IDSTA 2024, Dubrovnik, Croatia, September 24-27, 2024*, Mohammad A. Alsmirat, Yaser Jararweh, Moayad Aloqaily, and Haythem Bany Salameh (Eds.). IEEE, 89–94. doi:10.1109/IDSTA62194.2024.10747008

[3] Leonardo Guerreiro Azevedo, Ralf Hartmut Güting, Rafael Brand Rodrigues, Geraldo Zimbrão, and Jano Moreira de Souza. 2006. Filtering with raster signatures. In *14th ACM International Symposium on Geographic Information Systems, ACM-GIS 2006, November 10-11, 2006, Arlington, Virginia, USA, Proceedings*. ACM, 187–194. doi:10.1145/1183471.1183503

[4] Daniel Bahrdt and Stefan Funke. 2015. OSCAR: OpenStreetMap Planet at Your Fingertips via OSm Cell ARrangements. In *WISE (1) (Lecture Notes in Computer Science, Vol. 9418)*. Springer, 153–168. https://link.springer.com/chapter/10.1007/978-3-319-26190-4_11

[5] Jon Louis Bentley and Thomas Ottmann. 1979. Algorithms for Reporting and Counting Geometric Intersections. *IEEE Trans. Computers* 28, 9 (1979), 643–647. doi:10.1109/TC.1979.1675432

[6] Thomas Brinkhoff and Hans-Peter Kriegel. 1994. Approximations for a Multi-Step Processing of Spatial Joins. In *IGIS '94: Geographic Information Systems, International Workshop on Advanced Information Systems, Monte Verita, Ascona, Switzerland, February 28 - March 4, 1994, Proceedings (Lecture Notes in Computer Science, Vol. 884)*. Springer, 25–34. doi:10.1007/3-540-58795-0_31

[7] Timothy M. Chan. 1994. A Simple Trapezoid Sweep Algorithm for Reporting Red/Blue Segment Intersections. In *Proceedings of the 6th Canadian Conference on Computational Geometry, Saskatoon, SK, Canada, August 1994*. University of Saskatchewan, 263–268.

[8] Eliseo Clementini, Paolino Di Felice, and Peter van Oosterom. 1993. A Small Set of Formal Topological Relationships Suitable for End-User Interaction. In *Advances in Spatial Databases, Third International Symposium, SSD'93, Singapore, June 23-25, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 692)*, David J. Abel and Beng Chin Ooi (Eds.). Springer, 277–295. doi:10.1007/3-540-56869-7_16

[9] David H Douglas and Thomas K Peucker. 1973. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: the international journal for geographic information and geovisualization* 10, 2 (1973), 112–122.

[10] Francisco García-García, Antonio Corral, Luis Iribarne, and Michael Vassilakopoulos. 2023. Efficient distributed algorithms for distance join queries in spark-based spatial analytics systems. *Int. J. Gen. Syst.* 52, 3 (2023), 206–250. doi:10.1080/03081079.2023.2173750

[11] Liang Geng, Rubao Lee, and Xiaodong Zhang. 2024. RayJoin: Fast and Precise Spatial Join. In *Proceedings of the 38th ACM International Conference on Supercomputing, ICS 2024, Kyoto, Japan, June 4-7, 2024*, Kenji Kise, Valentina Salapura, Murali Annavaram, and Ana Lucia Varbanescu (Eds.). ACM, 124–136. doi:10.1145/3650200.3656610

[12] Thanasis Georgiadis and Nikos Mamoulis. 2023. Raster Intervals: An Approximation Technique for Polygon Intersection Joins. *Proc. ACM Manag. Data* 1, 1 (2023), 36:1–36:18. doi:10.1145/3588716

[13] Herring et al. 2011. OpenGIS Implementation Specification for Geographic information – Simple feature access – Part 1: Common Architecture. (2011). https://www.ogc.org/standard/sfa/

[14] Edwin H. Jacox and Hanan Samet. 2007. Spatial join techniques. *ACM Trans. Database Syst.* 32, 1 (2007), 7. http://www.cs.umd.edu/~hjs/pubs/jacoxtods07.pdf

[15] Isam Mashhour Al Jawarneh, Paolo Bellavista, Antonio Corradi, Luca Foschini, and Rebecca Montanari. 2024. SpatialSSJP: QoS-Aware Adaptive Approximate Stream-Static Spatial Join Processor. *IEEE Trans. Parallel Distributed Syst.* 35, 1 (2024), 73–88. doi:10.1109/TPDS.2023.3330669

[16] Andreas Kipf, Harald Lang, Varun Pandey, Raul Alexandru Persa, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2018. Adaptive Geospatial Joins for Modern Hardware. *CoRR* abs/1802.09488 (2018). arXiv:1802.09488 http://arxiv.org/abs/1802.09488

[17] Nikolaos Koutroumanis, Christos Doulkeridis, and Akrivi Vlachou. 2025. Parallel Spatial Join Processing with Adaptive Replication. In *Proceedings 28th International Conference on Extending Database Technology, EDBT 2025, Barcelona, Spain, March 25-28, 2025*, Alkis Simitsis, Bettina Kemme, Anna Queralt, Oscar Romero, and Petar Jovanovic (Eds.). OpenProceedings.org, 464–476. doi:10.48786/EDBT.2025.37

[18] Harry G. Mairson and Jorge Stolfi. 1988. Reporting and Counting Intersections Between Two Sets of Line Segments. In *Theoretical Foundations of Computer Graphics and CAD*, Rae A. Earnshaw (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 307–325.

[19] Sadegh Nobari, Farhan Tauheed, Thomas Heinis, Panagiotis Karras, Stéphane Bressan, and Anastasia Ailamaki. 2013. TOUCH: in-memory spatial join by hierarchical data-oriented partitioning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 701–712. doi:10.1145/2463676.2463700

[20] Wendy Osborn. 2021. Unbounded Spatial Data Stream Query Processing using Spatial Semijoins. *J. Ubiquitous Syst. Pervasive Networks* 15, 2 (2021), 33–41. doi:10.5383/JUSPN.15.02.005

[21] Varun Pandey, Andreas Kipf, Dimitri Vorona, Tobias Mühlbauer, Thomas Neumann, and Alfons Kemper. 2016. High-Performance Geospatial Analytics in HyPerSpace. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 2145–2148. doi:10.1145/2882903.2899412

[22] Jignesh M. Patel and David J. DeWitt. 1996. Partition Based Spatial-Merge Join. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, H. V. Jagadish and Inderpal Singh Mumick (Eds.). ACM Press, 259–270. doi:10.1145/233269.233338

[23] Franco P. Preparata and Michael Ian Shamos. 1985. *Computational Geometry - An Introduction*. Springer. doi:10.1007/978-1-4612-1098-6

[24] Urs Ramer. 1972. An iterative procedure for the polygonal approximation of plane curves. *Comput. Graph. Image Process.* 1, 3 (1972), 244–256. doi:10.1016/S0146-664X(72)80017-0

[25] Michael Ian Shamos and Dan Hoey. 1976. Geometric Intersection Problems. In *17th Annual Symposium on Foundations of Computer Science, Houston, Texas, USA, 25-27 October 1976*. IEEE Computer Society, 208–215. doi:10.1109/SFCS.1976.16

[26] M. Shimrat. 1962. Algorithm 112: Position of point relative to polygon. *Commun. ACM* 5, 8 (1962), 434. doi:10.1145/368637.368653

[27] Godfried T Toussaint. 1983. Solving geometric problems with the rotating calipers. In *Proc. IEEE Melecon*, Vol. 83. A10.

[28] Dimitrios Tsitsigkos, Konstantinos Lampropoulos, Panagiotis Bouros, Nikos Mamoulis, and Manolis Terrovitis. 2021. A Two-layer Partitioning for Non-point Spatial Data. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 1787–1798. doi:10.1109/ICDE51399.2021.00157

[29] Zhuohan Xu, Dejun Teng, Zhaohui Peng, and Fusheng Wang. 2024. Understanding the Intrinsic Characteristics of Spatial Partitioning in Distributed Spatial Join. In *IEEE International Conference on Big Data, BigData 2024, Washington, DC, USA, December 15-18, 2024*, Wei Ding, Chang-Tien Lu, Fusheng Wang, Liping Di, Kesheng Wu, Jun Huan, Raghu Nambiar, Jundong Li, Filip Ilievski, Ricardo Baeza-Yates, and Xiaohua Hu (Eds.). IEEE, 403–412. doi:10.1109/BIGDATA62323.2024.10825846

[30] Simin You, Jianting Zhang, and Le Gruenwald. 2015. Large-scale spatial join query processing in Cloud. In *31st IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2015, Seoul, South Korea, April 13-17, 2015*. IEEE Computer Society, 34–41. doi:10.1109/ICDEW.2015.7129541

[31] Simin You, Jianting Zhang, and Le Gruenwald. 2016. High-performance polyline intersection based spatial join on GPU-accelerated clusters. In *Proceedings of the 5th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data, BigSpatial@SIGSPATIAL 2016, Burlingame, California, USA, October 31, 2016*. ACM, 42–49. doi:10.1145/3006386.3006390

[32] Jianting Zhang and Simin You. 2012. Speeding up large-scale point-in-polygon test based spatial join on GPUs. In *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data, BigSpatial@SIGSPATIAL 2012, Redondo Beach, CA, USA, November 6, 2012*, Varun Chandola, Ranga Raju Vatsavai, and Chetan Gupta (Eds.). ACM, 23–32. doi:10.1145/2447481.2447485